About Lab 3
Lab 3 has a lot of scaffolding that is already implemented for you:

- There is a Square class that is completely implemented for you. Squares know their row and column in the maze, they know what type they are (open, wall, start or exit), you can mark them and set their previous square.
- There is Maze class that is completely implemented. This has a method loadMaze() that reads a maze from a textfile, and methods getStart() and getExit() to find the start and exit squares.
The Maze class also has as method `getNeighbors(Square sq)` that returns a list of the neighbors of `sq`. This includes all squares above, below, left, or right of `sq` even if they are walls or already marked.

There is a MazeSolver abstract class that is sketched out, but its important details are left for you to implement. MazeSolver has abstract methods for the worklist: `isEmpty()`, `Square next()`, `add(Square sq)` etc. Leave those as abstract.
You need to write two concrete methods of MazeSolver:

- **step( )** does one step of the algorithm in terms of the abstract worklist methods: If `isEmpty( )` says the worklist is empty there is no solution. Otherwise use `next( )` to get a Square from the worklist. Let’s call this square current. If current is the exit square you are done. If it isn’t then `maze.getNeighbors(current)` is a list of current’s neighbors. Mark those neighbors whose type isn’t WALL and who aren’t already marked, set their previous node to current, and add them to the worklist.
• step( ) should change MazeSolver’s variable pathFound to true when the exit node is found, and variable finished to true when either either the exit node is found or you are sure there is no solution.
• The other method of class MazeSolver that you need to write is `getPath()`, which returns an `ArrayList<Square>` that goes from the start square to the exit square.

• Finally, there are two concrete subclasses of MazeSolver that use specific implementations of the worklist. These are MazeSolverStack, which is completely implemented, and MazeSolverQueue which is not. If you read MazeSolverStack carefully you should see what you need to do for MazeSolverQueue.
So here is what you need to do for Lab 3:

a) Implement MyStack\(<E>\) using an ArrayList to hold the data. Test your implementation.

b) Implement MyQueue\(<E>\) using a linked structure to hold the data. Test your implementation.

c) In the MazeSolver abstract class you need to write methods step( ) and getPath( )

d) Implement MazeSolverQueue
You should then be able to run the MazeApp.
Here is the start of class MyStack:
public class MyStack<T> implements StackADT<T> {
    ArrayList<T> lister;
    public MyStack() {
        lister = new ArrayList<T>();
    }
    ...
}
Here is the start of class MyQueue:

```java
public class MyQueue<T> implements QueueADT<T>{
    class Node {
        T data;
        Node next;
        Node( T data) {
            this.data = data;
            this.next = null;
        }
        Node( ) {
            this(null);
        }
    }
    Node head, tail;
    public MyQueue( ) {
        head = new Node();
        tail = head;
    }
}```
Here is a picture to go with the MyQueue structure:

Both the stack and the queue structure throw a NoSuchElementException if you try to access an element of an empty structure.
Note that there are two ways to dequeue. Both require you to save `head.next.data` in a variable and return it at the end. For one version you remove the first element with `head.next = head.next.next`

Alternatively you remove it with `head = head.next`:

![Diagram showing dequeuing](image-url)
Both versions work correctly, but with the first version if you dequeue a queue with one element you need to explicitly set tail=head.